Million Songs Dataset – Report

DATA420 – Scalable Data Science

Yamika Gandhi – 43717539

BACKGROUND

The realm of music has witnessed a revolution in how songs are recommended to listeners and how genres are classified. Central to this revolution are innovative machine learning systems for song recommendations, aimed at delivering personalised music choices to the audience, and genre classification models, designed to identify and categorise the musical styles of diverse tracks.

The Million Song Dataset (MSD) is a very important resource in the field of music data analysis, stemming from an ambitious project initiated under the aegis of The Echo Nest, a research organisation born out of the MIT Media Lab. However, the Million Song Dataset is more than just a repository of music data; it's a community-driven initiative. Other organisations and individuals have also contributed additional datasets, like SecondHandSongs.com, musiXmatch.com, and last.fm.

After a decade of groundbreaking research, The Echo Nest was acquired by Spotify, a testament to the organisation's impact, at a staggering price of 50 million Euros.

This report explores the Million Songs Dataset to unlock its potential to create music recommendation systems and genre classification models.

DATA PROCESSING

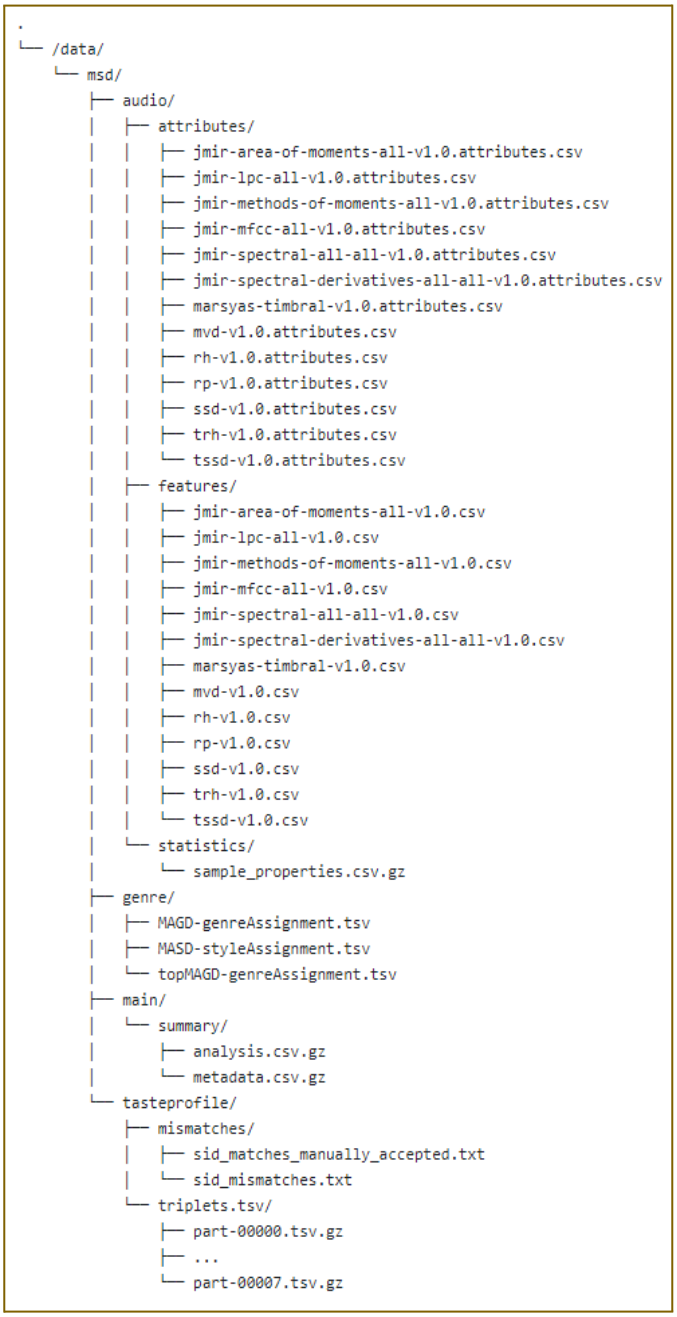
An overview of the datasets – some background, how they are stored in HDFS, directory tree, file sizes, and row counts.

The data has been provided to us via the Hadoop distributed file system. It is available locally on every node in the cluster. We can access it using the `'hdfs dfs'` commands.

The organization of the directories and files in HDFS is shown in the directory tree.

This can be obtained using the command `!hdfs dfs -ls "/data/msd"` and it gives us the following information-

There are 4 directories — `audio`, `genre`, `main`, and `tasteprofile`.



Audio Features - `msd/audio` — This dataset is obtained from Vienna University of Technology. They downloaded audio samples for **994,960** songs (in the Million Songs Dataset) as 30-60 second snippets. From these, they extracted various audio features like “rhythm histograms” and “timbral features”. These features allow us to compare songs and predict song attributes.

The “audio” directory contains 3 sub-directories, `attributes`, `features`, and `statistics`. Both “attributes” and “features” contain 13 CSV files each, with the same name. The CSV files in the “features” folder are actually directories, which contain the features data in the form of several CSV part-files. The files in the “attributes” folder contain the schemas (column headers and datatypes) for this data. For example, “rhythm histograms” is a feature (`msd-rh-v1.0`). This feature has its own dataset with 60 columns (components).

The “statistics” folder contains additional track statistics.

MSD AllMusic Genre & Style Data - `msd/genre` — The “genre” directory contains 3 TSV files. All 3 of these datasets put genre and style labels on track IDs, collected from Allmusic.com. There is the AllMusic Genre Dataset (MAGD), which contains 21 unique genres and has **422,714** rows, where each row represents a different track. The AllMusic Style Dataset (MASD) contains 25 different styles and **273,936** rows. And finally the Allmusic Top Genre Dataset (Top-MAGD) has a restricted set of genre labels, only 13 of them, because it excludes small genres such as *Religious* or *Christmas* and others, and also excludes non-musical content like *Comedy/Spoken*. It has **406,427** rows.

Main Dataset - `msd/main` — The "main" MSD dataset has audio analysis features (loudness, beat, tempo, time signature) and metadata (song ID, artist ID, year, title, artist tags). It has 51 fields in total. In the Hadoop system, it is given to us in 2 datasets – "`analysis.csv.gz`" and "`metadata.csv.gz`". Both the sets have **1,000,000** rows (for 1M songs) and **44,745** unique artists.

Taste Profile - `msd/tasteprofile` — This dataset has 1.2 million unique (and anonymous) listeners' taste profiles, and each taste profile has at least 10 MSD songs. This data has **384,546** unique MSD songs. It has already been matched to the MSD dataset, but unfortunately some songs have been mismatched.

It has 2 sub-directories, `mismatches` and `triplets`. The "mismatches" directory contains information about the mismatched songs. The "`sid_mismatches.txt`" file has **19,094** rows, which means that these are the songs that were automatically identified to be mismatched (predicted positives). These songs were then manually reviewed, and **938** songs were accepted to be not mismatched (false positives). These songs are listed in the "`sid_matches_manually_accepted.txt`" file. That leaves **18,156** songs that were actually mismatched (true positives). However, there are **1,000,000** unique songs in the main MSD dataset. The number of mismatched songs is insignificant compared to 1 million (less than 2%) and will not affect the collaborative filtering model that we will build later. Therefore, we **don't need to take any action** for this.

"`Triplets`" contains data about the play counts of a song in a triplet format, (user, song, play count), and these values are tab-delimited. One triplet is one user's taste profile. There are around **48 million** such taste profiles in the dataset.

Reading the audio features datasets

To read the audio features attributes data, I used the schemas in the `audio/attributes` directory. The attribute files and the feature datasets share the same prefix and are named consistently, so I **automated** the creation of `StructType` by **mapping attribute types** to `pyspark.sql.types` objects.

To read the audio features attributes data, I first read the schemas from the `attributes` directory. I put all the file names into a list called `filename`, and looped through it one by one, reading the schemas. They were given as 'ColumnName, ColumnType'. When the schema was read, the default names given to them were '`_c0`' and '`_c1`'. I used `.flatMap()` to extract column names from `_c0` and column types from `_c1`. I used a dictionary to map the column types to the appropriate attribute types in spark. I created an empty list called `struct_fields`. For each column name and its datatype, I created a `StructField` and appended it to the list. This gave me the schema. Then I stored the schema in a dictionary called '`schemas`' where the key was the `filename` and the value was the schema.

Then I read the schemas one by one from the dictionary and read the data into it from the `features` directory. For a step-by-step explanation of the code, please refer to the supplementary section of this report or view the supplementary code file.

The total size of data is obtained using the command `!hdfs dfs -du -h "/data/msd"` and the results are shown in Table 1, along with the row counts in each file.

How do these counts **compare to the total number of unique songs**?

There are 1,000,000 unique songs in the main dataset.

The MAGD genre dataset has 422,714 rows, so we have genre information for only about 422,714 songs. Similarly, we have style information for only 273,936 songs.

The audio features files contain 994,500 rows (on average). This means that audio features have been collected for roughly these many songs, with more features collected for some songs and less for others.

Table 1: File sizes, filenames, and respective row counts.

Size of 1 copy	Total size (of all copies)	Path of directory or sub-directory	Files	Row counts
103.0 KB	824.3 KB	data/msd/audio/attributes	msd-jmir-area-of-moments-all-v1.0.attributes.csv msd-jmir-lpc-all-v1.0.attributes.csv msd-jmir-methods-of-moments-all-v1.0.attributes.csv msd-jmir-mfcc-all-v1.0.attributes.csv msd-jmir-spectral-all-all-v1.0.attributes.csv msd-jmir-spectral-derivatives-all-all-v1.0.attributes.csv msd-marsyas-timbral-v1.0.attributes.csv msd-mvd-v1.0.attributes.csv msd-rh-v1.0.attributes.csv msd-rp-v1.0.attributes.csv msd-ssd-v1.0.attributes.csv msd-trh-v1.0.attributes.csv msd-tssd-v1.0.attributes.csv	21 21 11 27 17 17 125 421 61 1,441 169 421 1,177
12.2 GB	97.8 GB	data/msd/audio/features	msd-jmir-area-of-moments-all-v1.0 msd-jmir-lpc-all-v1.0 msd-jmir-methods-of-moments-all-v1.0 msd-jmir-mfcc-all-v1.0 msd-jmir-spectral-all-all-v1.0 msd-jmir-spectral-derivatives-all-all-v1.0 msd-marsyas-timbral-v1.0 msd-mvd-v1.0 msd-rh-v1.0 msd-rp-v1.0 msd-ssd-v1.0 msd-trh-v1.0 msd-tssd-v1.0	994,623 994,623 994,623 994,623 994,623 994,623 995,001 994,188 994,188 994,188 994,188 994,188 994,188
40.3 MB	322.1 MB	data/msd/audio/statistics	sample_properties.csv.gz	992,865
30.1 MB	241.0 MB	data/msd/genre	msd-MAGD-genreAssignment.tsv msd-MASD-styleAssignment.tsv msd-topMAGD-genreAssignment.tsv	422,714 273,936 406,427
174.4 MB	1.4 GB	data/msd/main/summary	analysis.csv.gz metadata.csv.gz	1,000,000 1,000,000
2.0 MB	16.2 MB	data/msd/tasteprofile/mismatches	sid_matches_manually_accepted.txt sid_mismatches.txt	938 19,094
488.4 MB	3.8 GB	data/msd/tasteprofile/triplets.tsv	part-00000.tsv.gz ... part-00007.tsv.gz	48,373,586

AUDIO SIMILARITY

This section is about trying to build a classification model which uses numerical representations of a song's audio waveform (obtained using methods such as digital signal processing and psycho-acoustic modeling) to predict its genre. The song's audio features are the predictors (or the x variables), and the genre is the class (or the y variable). We explore the Method of Moments dataset and the MAGD dataset for this. Such a classification model could help us compare songs based entirely on the way they sound.

Data exploration — summarising the statistics, identifying trends, and data visualisation

For the building of this model, I chose the **Method of Moments** features dataset, because it is the smallest one and has only 11 columns or features. The audio features in it are continuous, numerical values. I loaded this dataframe and named it `mm_df`. The MAGD data is called `magd_df`.

I renamed the column names to make them shorter and easier to type, for example, the column `Method_of_Moments_Overall_Standard_Deviation_1` became `mm_stddev_1`. Figure 1 shows the **descriptive statistics** for this dataset. As we can see, the standard deviations range from `0.066` to `2,089,356.436`. This means that all the features have different scales. Therefore, just by looking at the standard deviation, we cannot determine which feature has more variance and can give us more information.

index	count	mean	stddev	min	max
mm_stddev_1	994623	0.15498176001746336	0.06646213086143025	0.0	0.959
mm_stddev_2	994623	10.384550576952307	3.868001393874683	0.0	55.42
mm_stddev_3	994623	526.8139724398096	180.43775499775262	0.0	2919.0
mm_stddev_4	994623	35071.97543290272	12806.816272955562	0.0	407100.0
mm_stddev_5	994623	5297870.369577217	2089356.4364558063	0.0	4.657E7
mm_avg_1	994623	0.3508444432531317	0.18557956834383812	0.0	2.647
mm_avg_2	994623	27.46386798784071	8.352648595163766	0.0	117.0
mm_avg_3	994623	1495.8091812075547	505.89376391902306	0.0	5834.0
mm_avg_4	994623	143165.46163257837	50494.276171032274	-146300.0	452500.0
mm_avg_5	994623	2.396783048473542E7	9307340.299219666	0.0	9.477E7

Figure 1: descriptive statistics for all columns in "Method of Moments" audio-features data.

However, these statistics do tell us that we need to **normalise the data** so that all the feature values and the descriptive statistics come within the same range of [0,1].

Using normalised data will also give us a better logistic regression model. Normalisation ensures that all the features contribute more equally to the model. Otherwise, features with a large scale can overwhelm features with a small scale. Also, the coefficients assigned are also easier to understand and they truly reflect the importance of each feature, because now they are on a consistent scale.

So I started my process of transforming the data in `mm_df`. First, I used `VectorAssembler` and the `.transform()` method from `pyspark.ml` to combine all the (numerical) feature columns into a single feature vector column called "`features`". I saved this new dataframe as `assembled_mm_df`.

Then I normalised the data in `assembled_mm_df` using `MinMaxScaler` from the `pyspark.ml` library, which gave me another column called `scaled_features`. This column contains vectors of the normalised features, and this new dataframe was saved as `normalized_mm_df`.

After this, I used the normalised data to create a **correlation matrix** using the `.corr()` function, and visualised it as a heatmap using the `seaborn` library, as shown in Figure 2. This helped me find out which features are strongly correlated to each other.

Correlation indicates how strong the linear relationship is between 2 features. "Features with high correlation are more linearly dependent and hence have almost the same effect on the dependent variable. So, when two features have high correlation, we can drop one of the two features."

I am considering a correlation higher than **0.85** to be a **strong correlation**. The results are as follows -

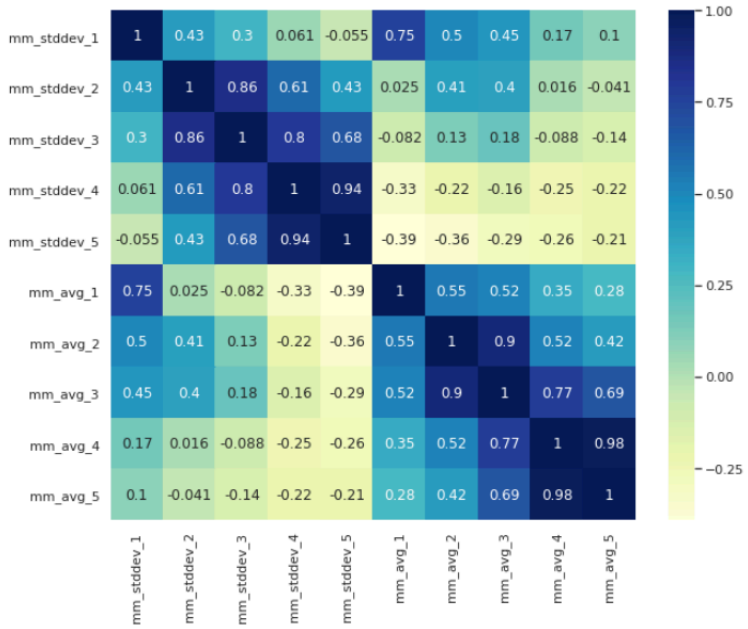


Figure 2: A correlation matrix for all the features in `mm_df`.

1. `mm_avg_4` and `mm_avg_5` have a correlation of 0.98
2. `mm_stddev_4` and `mm_stddev_5` have a correlation of 0.94
3. `mm_avg_2` and `mm_avg_3` have a correlation of 0.90
4. `mm_stddev_2` and `mm_stddev_3` have a correlation of 0.86
5. `mm_stddev_3` and `mm_stddev_4` have a correlation of 0.80

I **dropped the columns** `'mm_avg_4', 'mm_avg_2', 'mm_stddev_4'`, and then I recalculated `'features'`, and `'scaled_features'` for the remaining columns.

After this, I looked at the **MAGD dataset**, `magd_df`. It contains **422,714** unique tracks

I grouped the data by genre and counted the number of rows for each to understand the class distribution. The results are in Table 2, showing the counts by genre and the ratio of that genre in the data.

Around 56% of the songs are Pop Rock, followed by 9.7% Electronic, and 5% Rap.

A **visualisation** in the form of a bar chart is shown in Figure 3.

genre	count	ratio
Pop_Rock	238786	0.564888
Electronic	41075	0.097170
Rap	20939	0.049535
Jazz	17836	0.042194
Latin	17590	0.041612
RnB	14335	0.033912
International	14242	0.033692
Country	11772	0.027849
Religious	8814	0.020851
Reggae	6946	0.016432
Blues	6836	0.016172
Vocal	6195	0.014655
Folk	5865	0.013875
New Age	4010	0.009486
Comedy_Spoken	2067	0.004890
Stage	1614	0.003818
Easy_Listening	1545	0.003655
Avant_Garde	1014	0.002399
Classical	556	0.001315
Children	477	0.001128

Table 2: class distribution in MAGD.

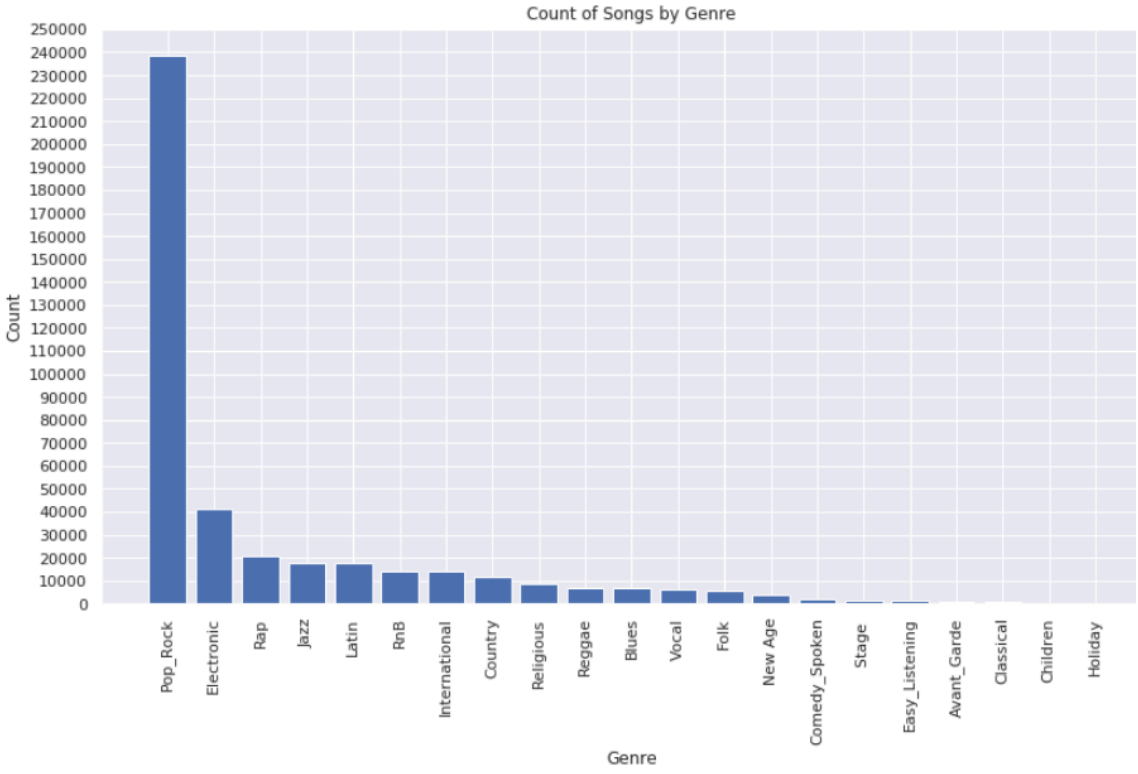


Figure 3: A bar chart showing the class distribution in MAGD.

Joining the Data and Preparing it for Training

The predictors (x variables) are present in `normalized_mm_df` while the class labels (y variable) are present in `magd_df`. So we must join these 2 in order to get a set of labelled data rows which can be used for classification.

Before doing this, I first checked a few things.

1. Unique track IDs in Methods of Moments audio-features dataset- 994,623
2. Unique track IDs in MAGD dataset- 422,714
3. Number of common track IDs in both datasets: 420,620
4. Number of track IDs present in `normalized_mm_df` but missing in `magd_df`: 574,003
5. Number of track IDs present in `magd_df` but missing in `normalized_mm_df`: 2,094

Initially, I joined the 2 datasets by using a left join, keeping `normalized_mm_df` on the left. But that resulted in a lot of null values in the `genre` column of the merged dataframe, because `normalized_mm_df` has 994,623 rows but it shares only 420,620 tracks in common with `magd_df`. This means that 574,003 tracks in the `normalized_mm_df` didn't have any `genre` label on them when joined with `magd_df`. Since the task here is to build a classification model, we need **labelled data** because classification is a supervised learning method. Therefore, the data which doesn't have any label is not of any use to us. To get rid of the rows where the `genre` was null, I went back and did an **inner join** instead, and saved the joined dataframe in `mm_and_genre_df`. The joined dataset now only includes the tracks that were common in both the datasets, and contains only labelled data.

Choosing the Algorithms for a Binary Classification Model

The next task was to develop a **binary classification model** which could predict whether a song belongs to the "Electronic" genre or not.

Now that the data was joined and labelled, there was one final step: **converting the 'genre' column into a binary column**, where 1 represents "Electronic" and 0 represents "Other".

I created a new column in `mm_and_genre_df` called `'label'` to represent this, using the function `F.when()`. I used the function `print_class_balance` to find the number of rows of the Electronic songs, and their ratio in the dataset. Table 3 summarises the class balance.

genre	label	count	class ratio
Other	0	379,954	90.3319%
Electronic	1	40,666	9.6681%

Table 3: class balance of the binary label.

Based on the descriptive statistics which I calculated above, I had already normalised the data and dropped the columns that were highly correlated to another column. So, all the **data processing was done** and now the data was ready for training the classification model.

I researched the classification algorithms available in the `spark.ml` library in order to choose **3 algorithms** I could use. I made my choice taking into account the interpretability, predictive accuracy, training speed, hyperparameter tuning, and several other factors.

Logistic Regression – This is the simplest classification algorithm and I thought it would be a good starting point, and it would be interesting to compare the results to those of more complex algorithms. This model is easy to interpret and to explain, and its coefficients indicate the importance of each feature. It performs well when the relationship between the features and the target variable is roughly linear but doesn't capture the complex patterns very well. It is fast to train. It has minimal hyperparameters which can be tuned easily. It can work well with high-dimensional data, however, in our case there shouldn't be an issue because after dropping some columns, we only have 7 features left in our data.

Random Forest – I chose this because I wanted to explore decision trees. Random forest is an ensemble method that uses multiple decision trees, and it is an improvement, so I chose random forest instead. While a decision tree is easy to interpret, an ensemble of trees can be different to understand, especially if the trees are deep. Trees can capture complex relationships and it can be difficult to follow how a prediction was made, unlike logistic regression where coefficients have a clear meaning. Feature importance is evaluated by measuring how much a feature reduces the impurity of the classification, which is measured by the Gini Index. Random forest has high predictive accuracy because it can handle non-linear relationships and noisy data, so its results should be quite different from logistic regression. However, training this algorithm is slow because of a large number of trees. Some of its hyperparameters are the number of trees, their depth, the number of features to consider while making a split, etc. It can handle high-dimensional data and it's not sensitive to feature scaling.

Neural Network (multilayer perceptron classifier) – I chose this because this is very different from the 2 above mentioned algorithms. I also haven't had a chance to explore neural networks much and I wanted to take this opportunity. Neural networks give pretty accurate predictions and can handle complex relationships. Training neural networks can be time consuming. They can handle high-dimensional data but they are sensitive to feature scaling, which shouldn't be a problem because our data is normalised. They require large amounts of data to train.

Creating the Train and Test Sets Via Stratified Random Sampling

Before building a classification model, I split my data in `mm_and_genre_df` into **training and test sets**.

A **class imbalance** makes it difficult to train and evaluate the performance of a model. The accuracy can be misleading, and the model might not generalise well to new data. Therefore, we need to take care of certain things when we split our data.

The **test** data must have the same class balance as the actual data, so that it provides a fair representation of how the model would perform on new data (which would have the same class balance).
However, we have a little more freedom with what we have in our **training** data. We can choose to have more of the rare class (upsampling) or less of the common class (downsampling).

The ideal way to preserve the class balance in our training and test sets is to do **stratified random sampling**. This method divides the entire data into mutually exclusive classes or strata (using the `Window` function of `pyspark.sql`). Then, we randomly sample observations from each stratum/class. We can choose how many observations we want to sample from each class. Here, I've put 80% of the data in the training set and 20% in the test set. Table 4 shows the composition of the training and test sets.

The proportion of the positive class 1 in the data is quite small compared to class 0. So I chose to do **downsampling** to remove some examples of class 0 from the **training** data.
This is because when we have too many examples of a class, there might be some redundant information there as well, which can be removed from the training data without really affecting the model performance. I decided to remove 50% rows from class 0, which left only **151,962** rows of class 0 in the training data as compared to the previous 336,495. The number of class 1 examples stayed the same. This resulted in a class ratio of **82.3635%** of class 0 and **17.6192%** of class 1. This was good enough. The balance didn't need to be 50-50 because that would have made the model too sensitive, and it could have started to underperform on the class 0.

dataset	total rows	genre	label	count	class ratio
training	336,495 (80% of the data)	Other	0	303,963	90.3321%
		Electronic	1	32,532	9.6679%
test	84,125 (20% of the data)	Other	0	75,991	90.3311%
		Electronic	1	8,134	9.6689%

Table 4: composition of training and test sets after stratified sampling

The downsampling was done by creating a new column called 'random' which contained random numbers from 0 to 1. This acted as an 'index' for rows. I kept all the rows in class 1, and kept only those rows from class 0 for which the value in the 'random' column was less than 0.5, which removed 50% of the rows from class 0. I saved my data as `test` and `training_downsampled`.

I didn't do observation weighting because deciding the weights for a class requires domain knowledge.

Training the Binary Classification Models

I trained all the chosen algorithms using the `pyspark.ml.classification` module and the `.fit()` function. I imported the classes `LogisticRegression`, `MultilayerPerceptronClassifier`, and `RandomForestClassifier`. I passed the `scaled_features` column and the `label` columns to them for training from my data `training_downsampled`. In neural networks, the "layer" argument is required. I assigned 5 neurons to the hidden layer. The other 2 algorithms have default values for their hyperparameters.

Evaluating the Model Performance on the Test Set

I used the `.transform()` function to fit the model on the test data and used the user-defined functions `print_metrics()` and `with_custom_prediction()` to compute and **print performance metrics** such as range, accuracy, and recall/sensitivity.

Precision tells us: how often are the positive predictions correct?
Recall tells us: can an ML model find all instances of the positive class?
Accuracy shows how often a classification ML model is correct overall.

The performance metrics I got were not too good. Table 5 shows a summary of the results I got. I first calculated my metrics with a **threshold of 0.5**. Random forest gave me the best values for precision, accuracy, and AUROC. In all the 3 models, accuracy was quite good, but this could be because of the class imbalance that was in the training data even after downsampling.

model	threshold	precision	recall	accuracy	AUROC
Logistic regression	0.5	0.3719	0.0508	0.8999	0.6944
Random forest	0.5	0.4877	0.1075	0.9027	0.7537
Neural network	0.5	0.3922	0.1101	0.8974	0.7466
Logistic regression	0.3	0.2724	0.2833	0.8575	0.6944
Random forest	0.3	0.3714	0.3149	0.8822	0.7538
Neural network	0.3	0.2678	0.4479	0.8282	0.7466
Logistic regression	0.7	0.2227	0.0052	0.9020	0.6944
Random forest	0.7	0.0000	0.0000	0.9033	0.7538
Neural network	0.7	0.3235	0.0040	0.9028	0.7466

Table 5: performance metrics for all algorithms with different thresholds

It's hard to decide which metric to optimise, precision or recall. Usually we decide this based on which has a higher cost – false positives or false negatives. However, this is just a song-classification system, and a wrong classification will not have a high cost as such. Balancing precision and recall depends on the **decision threshold** that we use for classification.

Finding the optimal threshold is outside the scope of this report, but I tried 2 different values of thresholds to see if the performance would change.

A threshold of 0.3 gave much better results, while a threshold of 0.7 gave terrible metrics. Therefore, we see that changing the threshold can sometimes improve classification performance when we have a class imbalance.

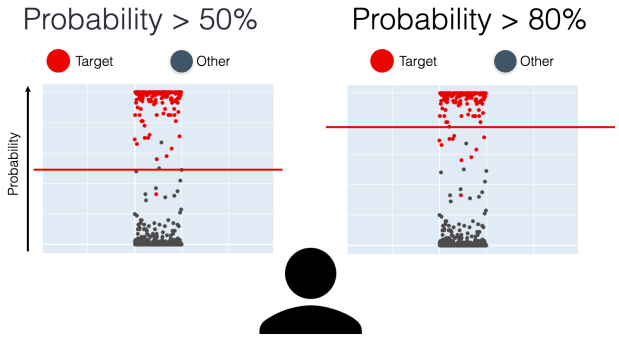


Figure 4: classification threshold (evidentlyai.com)

Hyperparameter Tuning and Cross-Validation

Model parameters are the internal variables learned by the machine learning algorithm during the training process from the data. They define the model's relationship between input data and the target output. They are not set manually.

The hyperparameters are external configuration settings for a machine learning algorithm and are set before the training based on domain knowledge. They directly affect a model's learning process and therefore they must be tuned for good results.

While training, the models were given default hyperparameters. The only one I set was the number of neurons in the hidden layer for my neural network. I looked up the hyperparameters using `.extractParamMap()`, which gave me a dictionary where the key was the hyperparameter name and the value was the value of the hyperparameter. This gave me all the hyperparameters. Table 6 provides a summary of the default hyperparameters.

Briefly, here is my understanding about how these hyperparameters affect the mode and its performance:

Logistic Regression – `elasticNetParam` combines lasso and ridge regularisation. `maxIter` is the maximum number of iterations for optimization. `regParam` is the regularisation strength. `threshold` is the probability value above which the model classifies the instance as the positive class. `standardisation` scales the features if set to True.

Random Forest – `maxDepth` is the maximum depth of the trees in the forest. `numTrees` is the number of decision trees in the forest. `subsamplingRate` is the fraction of the training data used for building each tree. `impurity` is the measure of impurity used.

Neural Network – `layers` defines the architecture of a neural network. `maxIter` is the maximum number of iterations for optimization. `stepSize` is the learning rate which controls how quickly the parameters are updated during training. `solver` is the optimization algorithm.

Logistic regression		Random forest		Neural network	
elasticNetParam	0	bootstrap	True	maxIter	100
maxIter	100	impurity	Gini	solver	1-bfgs
regParam	0	maxDepth	5	stepSize	0.03
standardisation	True	minInfoGain	0	layers	[7,5,2]
threshold	0.5	numTrees	20		
		subsamplingRate	1.0		

Table 6: Default hyperparameters for all the models.

Cross-validation is a resampling technique used to validate the performance of a model. It is helpful for tuning hyperparameters because it eliminates the problem of **data leakage**.

Data leakage happens when information from outside the training dataset is used to create a model, and this additional information can influence a model's performance.

In traditional machine learning workflows, we divide our data into a training set and a test set, which is important for evaluating a model's performance. However, during hyperparameter tuning, we can't solely rely on the training data for decision-making, and we also can't evaluate the model on the test data until our choices about model parameters are finalised. The test data should be reserved for a final, unbiased evaluation.

Therefore, we use the concept of 'validation', which is done by creating a **validation dataset** as a subset of the training data. This validation set can be used for evaluation of the model while the training is still going on and for tuning the hyperparameters. However, if we make all our decisions about the hyperparameters on the validation data, there is a risk of overfitting on the validation set. So to evaluate our model during training and to get a better estimate of how well it will generalise to new data, we do cross-validation. Here is how it works:

1. We partition the training data into k segments or "folds" of equal sizes.
2. We have k iterations of training.
3. For each iteration, one of the folds is used for validation and the rest are used for training using certain hyperparameters.
4. The trained model is used to make predictions on the validation set.
5. The performance metrics are recorded for each iteration, and then aggregated at the end.

6. We run multiple cross-validation iterations with different hyperparameters, and then choose the set of hyperparameters that results in the best model performance.

For a general classification algorithm, if we want to tune the hyperparameters, we need to evaluate the performance of a model for multiple combinations of hyperparameters. For this we need an optimization technique that allows us to explore the parameter space. Defining a **hyperparameter grid** is one such method. To do this, I would first identify the hyperparameters that significantly influence the algorithm. Then I would create a set of potential values for each hyperparameter. I would do a grid search, which involves exhaustively exploring all possible combinations of hyperparameters. A grid can be created by using the `ParamGridBuilder` module from `pyspark.ml.tuning`. This method is systematic but can be computationally expensive, especially if the grid is large. I would do cross-validation for each combination of the hyperparameters. Then I would calculate the average performance metric across all iterations to get an estimate about how a combination of hyperparameters is performing. The best performing combination of hyperparameters would be selected.

How much I expect hyperparameter tuning to improve the performance metrics is quite subjective, and depends on a lot of things. If the default hyperparameter values are well-suited to the problem, then tuning would provide only marginal improvements. Perhaps tuning is more useful for more complex models than the simpler ones.

Developing a Multiclass Classification Model

I chose **Random Forest** for this because it is capable of multiclass classification and can handle class imbalance.

I copied the data from `mm_and_genre_df` to a variable called `data_for_multiclass` to avoid any naming mix-up with the previous classification.

For multi class classification, I first converted the genre column in `data_for_multiclass` into an integer index that **encodes** each genre consistently. This new column was named `genre_index` and I did this using the `StringIndexer` class from `pyspark.ml.feature`. It's used for converting categorical text data, such as strings, into numerical values as a part of preprocessing. Doing this required minimum work by hand.

Then I split `data_for_multiclass` into train and test. I did this using stratified sampling, similar to how I did it for binary classification. Using `Window.partitionBy("genre_index")`, I divided the data into separate classes and took 80% data from each class to create `train_mc`. Then I did a left anti-join to put the remaining rows into `test_mc`, where 'mc' stands for multiclass. This gave me training and test data which preserved the class imbalance of the original data. I had **336,483** rows in `train_mc` and **84,137** rows in `test_mc`. I printed out the class balance in my train and test data along with the class ratios to confirm that the imbalance was preserved, and I saw that it was.

Then I trained my Random Forest model on the imbalanced training data, fit it on the test data, and printed out the metrics.

Since I wanted to compare how the class balance affects a model's performance, I also **resampled the training data** to balance the classes slightly, and trained the model on the resampled data.

For resampling, I used the function `random_resample_udf`. This function looks at the total count of each genre in the `train_mc` data. If that count is greater than `count_upper_bound`, then it undersamples that genre to make the count equal to upper bound, and if the count is lower than `count_lower_bound`, then it oversamples that genre to make the count equal to lower bound. These bounds are set manually and I set the lower

bound as 2,000 and the upper bound as 50,000. I saved the resampled data in `train_mc_resampled`, which now had **204,718** rows. Only the genres which were outside the bounds got resampled and their row counts changed. The other genres had the same row counts as before. For example, Pop_Rock initially had 190,119 rows in `train_mc`, but got downsampled to 50,240. However, Electronic had 32,532 rows initially and this remained unchanged.

So, for multiclass classification, I created 2 Random Forest models: one was trained on the imbalance data and was called `rf_mc_imbalanced_model`, and the other was trained on the resampled data and was called `rf_mc_resampled_model`.

Initially I trained my models with the default hyperparameters, which means a `maxDepth` of 5. However, for multiclass classification, we must have at least as many leaf nodes as there are classes. Since the depth of a tree is directly related to the number of leaf nodes, I increased the depth to 15, and I found that the models performed better. It took a long time to train the models at this depth.

I fit both the models on `test_mc`. The test data preserved the imbalance of the original data, because the test data is supposed to emulate the data in the real world.I then printed the metrics using `MulticlassClassificationEvaluator`. The metrics **relevant to multiclass classification** are weighted precision, weighted recall, and accuracy. The performance of the 2 models is shown in Table 7. The metrics are almost the same, which shows that the **class balance** in the training data **does not affect** the performance of the model.

Random Forest on Imbalanced Data		**Random forest on Resampled Data**	
Weighted precision	0.4996	Weighted precision	0.4817
Weighted recall	0.5878	Weighted recall	0.5250
Accuracy	0.5878	Accuracy	0.5250
F-1 score	0.4799	F-1 score	0.4902

Table 7: Performance of random forest models on multiclass classification

Next, I wanted to see how the performance was affected by the **inclusion of multiple genres**. To do this, I compared the performance of Random Forest on binary data and on multiclass data.

The model `rf_mc_resampled_model` was trained on resampled data, in which the classes were balanced slightly. It was trained on the default hyperparameters except for `maxDepth`, which was 15.

Random Forest on Binary Data (rf_bc_model)		**Random forest on Multiclass Data** (rf_mc_resampled_model)	
Precision	0.4878	Weighted precision	0.4817
Recall	0.2943	Weighted recall	0.5250
Accuracy	0.9018	Accuracy	0.5250
AUROC	0.8135	F-1 score	0.4902

Table 8: Performance of random forest models on binary vs multiclass

To make a fair comparison, it's important to keep everything the same. So I trained a new Random Forest model on `training_downsampled` (which has binary labels of 0 and 1), and again set the `maxDepth` to 15. This model was named `rf_bc_model`. Table 8 shows the performance metrics.

The precision in both was almost the same. The recall improved in multiclass classification. The accuracy was much higher in binary classification.

SONG RECOMMENDATIONS

In this section, I used the Taste Profile dataset to develop a song recommendation service based on **collaborative filtering**, which involves generating song recommendations for specific users based on the combined **user-song-play** information from all users.

Data exploration — partitioning the data, getting an overview, and data visualisation

I read the `tasteprofile.tsv` file, renamed the columns, and saved the data in `triplets_df`.

This data has **48,373,586** rows, each of which is a triplet of (`user_id, song_id, play_count`). This is a lot of data, and for faster processing, I decided to partition and cache my data. I determined the ideal number of partitions by looking at my configuration using `.getConf()`, and segregated my data into **96** partitions.

This dataset has **384,546** unique songs and **1,019,318** unique users.

There are 2 ways to interpret the **most active user** in the dataset.

The first approach is that the most active user is someone who has listened to the greatest number of distinct tracks, which might indicate that this user spends a lot of time on the song-streaming app searching for different songs. With this perspective, the most active user is "Ec6dfcf19485cb011e0b22637075037aae34cf26" and has played **4,400** distinct songs, which is **1.144%** of the unique songs in the dataset.

The second approach is that the most active user is someone who has the highest number of play counts in total, because it's likely that this user has spent the most amount of time on the song-streaming app. So I grouped the data by `user_id` and summed up the total plays for that user. The most active user had a total play count of **13,132** and had the `user_id` "093cb74eb3c517c5179ae24caf0ebec51b24d2a2". This user played **202** different songs, and this was **0.0525%** of the dataset. It was surprising to find that there is one particular song, "SOAOSDF12A58A779F1", which this user has played 9,667 times.
It was even more surprising to find that when I ran commands like `.describe()` or `F.max()`, this particular outlier was not detected and the highest play count was reported to be 995. I decided to drop the row with the outlier because it would have interfered with the descriptive statistics that I wanted to calculate.

I visualised the distribution of **song popularity** by collecting the counts of user plays per song into a Pandas dataframe, and plotted a histogram. I also plotted a histogram of **user activity** by collecting the counts of song plays per user. The results can be seen in Figure 4. These graphs follow an **exponential distribution**.

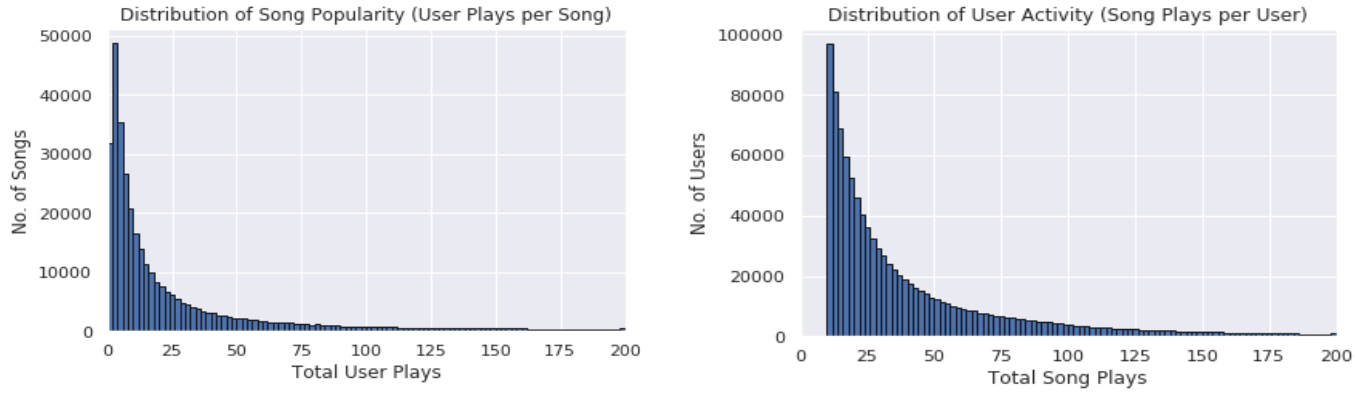


Figure 5: visualising song popularity and user activity.

Preparing the Data for a Collaborative Filtering Model

Collaborative filtering determines similar users and songs based on their combined play history. Songs which have been played only a few times and users who have only listened to a few songs will not contribute much information and are unlikely to improve the model. Therefore, I cleaned my dataset by removing songs which have been played less than **N=100 times**, and users who have listened to fewer than **M=20 songs**. I decided this based on the following descriptive statistics that I calculated –

1. How many songs does a user listen to on average? - 47.4568
2. How many listeners does a song have on average? - 125.794
3. How many times is a song played on average? - 360.608
4. How many times does a user play any song on average? - 136.042

I filtered the songs based on N and saved it as `filtered_songs`, and also filtered the users based on M and saved it as `filtered_users`, and then inner-joined both the tables and called it `cleaned_triplets_df`. This left me with **38,158,181** rows.

Then I used `StringIndexer` to encode the `user_id` and `song_id` into numeric values and created `user_index` and `song_index`, because such an encoding is needed before training the ALS model.

Then I split the data into `train_cf` and `test_cf`. With collaborative filtering, it is very important that the data that we have in the test set must also be present in the training data, because recommendations are made based on the patterns observed in the user's historical interactions with items (e.g., songs). The collaborative filtering models rely on learning user and item embeddings, or latent factors from observed user-item interactions in the training data. If a user doesn't have any interactions in the training set, it's challenging for the model to make accurate recommendations for them.

To create a **75-25 train-test split**, I first added a column called "`rand`" using `F.rand()` which had a random number between 0 and 1, and used this column to shuffle the whole dataset. I used `Window.partitionBy("user_id")` to group the data by user, so that from each group I could put 25% data into `test_cf` and 75% into `train_cf`. I assigned row numbers (`row_num`) to each row within a user group, and also calculated the maximum number of rows (`max_row_num`) in that user group. After this, I created a new column called "`test_threshold`" which was calculated by multiplying (`max_row_num*0.25`). If the `row_num` was less than `test_threshold`, the interaction went into `test_cf`, otherwise it went into `train_cf`. This ensures that every user in the test set is also in the train set, while keeping the selection random.

The train data has **28,537,589** rows and the test data has **9,620,592** rows.

Training and Testing the Collaborative Filtering Model using ALS

I used the `spark.ml.recommendation` library to train an implicit matrix factorization model using ALS on `train_cf` with the settings `maxIter=5`, `regParam=0.01`, and `implicitPrefs = True`. I named it `alsModel`.

I randomly picked 5 users, and stored their `user_index` in a list, and used it to filter out the rows of `test_cf` data. I saved this data in `handpicked_users_data`. I generated 5 recommendations for the users using `alsModel.recommendForUserSubset(handpicked_users_data, 5)` and saved them in

`recommendations_handpicked`. Then, to compare these with what the user had actually listened to, I collected and sorted the songs they had listened to, and saved them in `relevant_handpicked`. I merged these 2 dataframes on `user_index` so that I could view the `song_index` side by side. As we can see in Table 9, not a single recommendation matches what they have listened to. Therefore, this does not seem to be a good model. Finally, I tested the model on the full data. I generated 5 recommendations each for all users and saved them as `recommendations_all`. Then, just like before, I collected and sorted the songs that the users in the `test_cf` data had *actually* listened to, and saved this as `relevant`. Then I merged the recommendations and relevant items so that they could be compared.

These are the metrics for the collaborative filtering model–

- Precision@10 : 0.02652
- MAP@10 : 0.01887
- NDCG@K : 0.04216

user_index	recommendations_handpicked	relevant_handpicked
472363	[203.0, 98.0, 1.0, 88.0, 382.0]	[21516.0, 25690.0, 17990.0, 11592.0, 7872.0, 2...
541958	[0.0, 10.0, 20.0, 30.0, 40.0]	[41553.0, 20289.0, 18051.0, 9186.0, 3810.0, 35...
403878	[0.0, 3.0, 7.0, 6.0, 11.0]	[84.0, 54.0, 370.0, 61.0, 5380.0, 4099.0, 3031...
583742	[14.0, 203.0, 1.0, 382.0, 5.0]	[19648.0, 14662.0, 10106.0, 6567.0, 4676.0]
123456	[2.0, 4.0, 9.0, 5.0, 30.0]	[12212.0, 8.0, 254.0, 94.0, 21.0, 65105.0, 336...

Table 9: song recommendations and actual songs played, side by side.

Precision@K – it measures the proportion of relevant items among the top-K recommended items. It assesses how well the system ranks relevant items at the top. However, it only considers the top-K recommendations and ignores other relevant items. It also doesn't consider the ordering of recommendations beyond the top-K.

NDCG @ K (Normalised Discounted Cumulative Gain @ K) – NDCG is a measure of ranking quality that considers the position of relevant items in the recommendation list. It rewards higher-ranking of relevant items in the recommendation list.

MAP (Mean Average Precision) – It calculates the average precision for each user and then computes the mean. Precision at each relevant item's rank is used, and it rewards precision across the entire recommendation list. It is useful for evaluating the system's performance across all users. However, it may not capture the user's behaviour when they only look at the top recommendations.

Limitations of these metrics in evaluating recommendation services or collaborative filtering models:

- They focus on the top-K recommendations and may not reflect user satisfaction with long-tail items.
- They don't account for user diversity, which is crucial in real-world recommendation systems.
- They assume that users examine recommendations sequentially and might not capture user behaviour accurately.

A/B Testing can be an **alternate method** for comparing two recommendation services. It involves testing different recommendation algorithms on real users. Metrics like click-through rate, conversion rate, and user engagement are used to evaluate the impact of recommendations on actual user behaviour.

Additional metrics for measuring recommendation services would be user engagement (liking a song or adding it to a playlist), click-through-rate, conversion rate, and time spent listening to a song.

CONCLUSION

In this assignment, I worked with a large song dataset using HDFS and Spark. I processed the data using pyspark, and preprocessed the data, performed analysis to draw insights, and answered the questions asked.

I explored the dataset, calculated descriptive statistics and provided an overview. I discussed the structure and format of our data..

Next, I built a binary and a multiclass classifier to predict song genres based on audio features of songs. I chose the method of moments audio feature for this. I briefly discussed hyperparameter tuning.

In the last section, I built a song recommendation system based on the taste profiles of more than 1 million users, based on collaborative filtering.

This assignment gave me the space to explore the concepts in a practical way and to apply the lecture material to an actual dataset.

However, due to exams approaching, I was unable to give much time to this assignment. I would have liked to explore the collaborative filtering part a little more, and figure out why the model didn't work well.

REFERENCES

1. Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. *The Million Song Dataset*. In Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR 2011), 2011.
2. http://millionsongdataset.com/
3. Vishal R. "Feature Selection — Correlation and P-value." Towards Data Science, 12 Sep 2018, https://towardsdatascience.com/feature-selection-correlation-and-p-value-da8921bfb3cf.
4. ChatGPT – "Explain the difference between model parameters and hyperparameters."
5. How to use classification threshold to balance precision and recall (evidentlyai.com)
6. Ronaghan, Stacey. "The Mathematics of Decision Trees, Random Forest and Feature Importance in Scikit-learn and Spark." Towards Data Science, 12 May 2018, https://towardsdatascience.com/the-mathematics-of-decision-trees-random-forest-and-feature-importance-in-scikit-learn-and-spark-f2861df67e3
7. Özen, Burak. "Multi-class Classification on Imbalanced Data using Random Forest Algorithm in Spark." Medium, 19 Jan 2021, https://burakozen.medium.com/multi-class-classification-on-imbalanced-data-using-random-forest-algorithm-in-spark-5b3d0af9b93f

SUPPLEMENTARY

This can be done using a for-loop. Here is how this works-

1. I've created a list of file names called `filenames`. This can be used for 'attributes' as well as 'features', since they both use the same file names.
2. The filenames will be plugged into the base path to construct paths for all files.
3. A dictionary called `schemas` has been created to store all the schemas that will be read.
4. A file can be read using `spark.read.csv`. This gives us a dataframe which is stored in the local variable `attributes_df`. This dataframe has 2 columns (named in a default way by spark), which are `_c0` and `_c1`.
5. `_c0` contains information about the column header. Using `flatMap` and `.collect()`, we can extract the column names and turn them into a python list.
6. Similarly, `_c1` contains information about the datatype. We can return these as a list as well.
7. Then we create a `type_mapping` dictionary which will map attribute types like 'real' and 'string' to corresponding PySpark data types.
8. We create a list called `struct_fields` which will store all the `StructField`s that we create. A struct_field is a field/column in a `StructType` structure.
9. We run another for-loop which creates a `StructField` for a column name, and maps it to the correct datatype using the `type_mapping` dictionary. We add each StructField to the list we created.
10. The list `struct_fields` is passed to `StructType` to create a schema for each filename.
11. This schema is stored in the dictionary `schemas`, where the key is the filename, and the value is the schema.

7]:
```
filenames = [
    "msd-jmir-area-of-moments-all-v1.0",
    "msd-jmir-lpc-all-v1.0",
    "msd-jmir-methods-of-moments-all-v1.0",
    "msd-jmir-mfcc-all-v1.0",
    "msd-jmir-spectral-all-all-v1.0",
    "msd-jmir-spectral-derivatives-all-all-v1.0",
    "msd-marsyas-timbral-v1.0",
    "msd-mvd-v1.0",
    "msd-rh-v1.0",
    "msd-rp-v1.0",
    "msd-ssd-v1.0",
    "msd-trh-v1.0",
    "msd-tssd-v1.0"
    ]
```

```
import os

# creating a dictionary to store attribute schemas
schemas = {}

# looping through filenames and creating schemas
for filename in filenames:

    attributes_path = f"hdfs:///data/msd/audio/attributes/{filename}.attributes.csv"
    attributes_df = spark.read.csv(attributes_path, header=False, inferSchema=True)

    # Extract column names from the attributes DataFrame
    column_names = attributes_df.select("_c0").rdd.flatMap(lambda x: x).collect()

    # Extract data types from the attributes DataFrame
    column_types = attributes_df.select("_c1").rdd.flatMap(lambda x: x).collect()

    # a dictionary to map attribute types
    type_mapping = {
        'real': DoubleType(),
        'string': StringType(),

    }

    struct_fields = []

    for name, data_type in zip(column_names, column_types):

        # creating a StructField object and append it to the struct_fields list
        struct_field = StructField(name, type_mapping.get(data_type,StringType()), True)
        struct_fields.append(struct_field)
        schema = StructType(struct_fields)


    # creating the StructType and store it in the dictionary
    schemas[filename] = schema
```

Now the next step is to match the schemas to the datasets in the 'features' directory. This is done as follows-

(here, the 'filename' means a directory which has data in the form of several csv part-files.)

1. We create an empty dictionary `feature_dataframes` to store dataframes for features.
2. We loop through the filenames and perform these actions for each file-
3. We access the correct schema from `schemas` using the filename.
4. We construct a `features_path` for that filename.
5. We use `spark.read.csv` to read data from these part-files, which is returned to us as a dataframe `feature_files_df`.
6. We extract the names of the columns into a list called `existing_columns`.
7. Then we loop over the indices in the selected schema, use that index to pick an existing column, then pick the corresponding column name from 'schema', and rename that column. We store all the renamed columns in a new list called `renamed_columns`.
8. We replace the columns in `feature_files_df` with the renamed columns.
9. Now this modified dataframe can be stored in the dictionary `feature_dataframes` where the key is the filename and the value is the modified dataframe. We can access the data from here.

(This was written with the help of ChatGPT)

```
from pyspark.sql.functions import col


# Create a dictionary to store DataFrames for features
feature_dataframes = {}

# Iterate through feature directories and read and structure the CSV files

for filename in filenames:

    schema = schemas[filename]

    features_path = f"hdfs:///data/msd/audio/features/{filename}.csv"

    # this reads all the CSV part-files within a features directory and returns a dataframe
    feature_files_df = spark.read.csv(features_path, header=False, inferSchema=True)

    # extracting the existing column names from the DataFrame
    existing_columns = feature_files_df.columns

    # Rename columns in the DataFrame to match the schema, and retain extra columns
   # Create an empty list to store the renamed columns
    renamed_columns = []

    # Loop through the indices of the schema fields
    for i in range(len(schema)):
        # Get the existing column name at the corresponding index
        column = col(existing_columns[i])

        # Get the new field name from the schema for the corresponding field index
        new_field_name = schema.fieldNames()[i]
        # Alias the Column with the new field name
        renamed_column = column.alias(new_field_name)
        # Append the renamed column to the list
        renamed_columns.append(renamed_column)

    # replacing the columns in the DataFrame (feature_files_df) with the renamed columns.
    feature_files_df = feature_files_df.select(*renamed_columns)

    # Store the DataFrame in the dictionary with the filename/directory name as the key
    feature_dataframes[filename] = feature_files_df
```